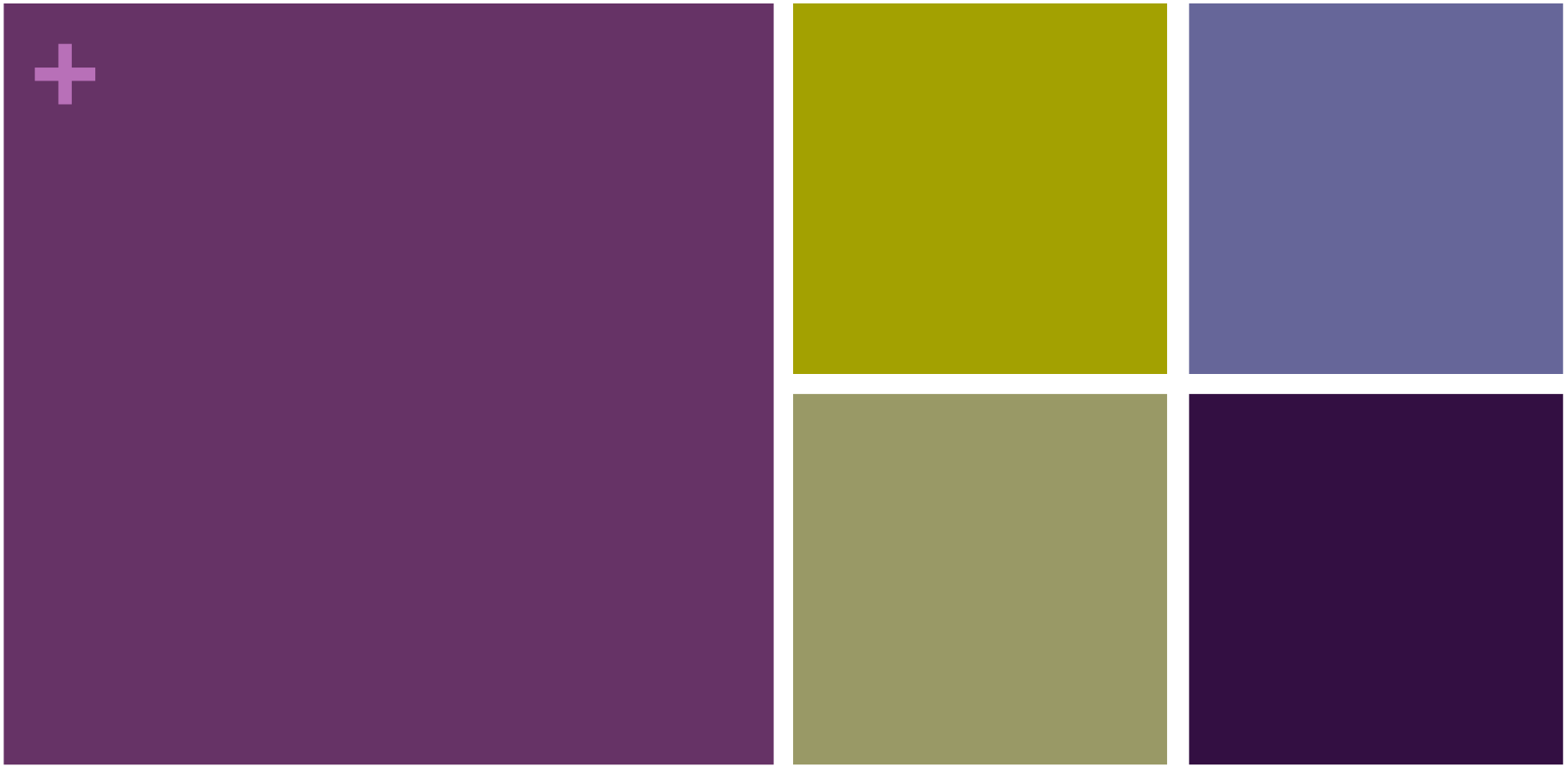


Sistemas Distribuidos
REST

Rodrigo Santamaría

- REST
  - Uso de un servicio web
    - Autenticación
    - Mediante URIs
    - Mediante una aplicación Java
    - Parseo de XMLs
  - Creación de un servicio web en Java

Uso de un servicio web

Servicios

- Existen muchos servicios web cuya API se puede utilizar (generalmente, previa autenticación)
- Una buena colección de apis públicas:
  - https://github.com/public-apis/public-apis
- Algunos ejemplos que no necesitan autenticación:
  - Agencia Estatal de Meteorología (Aemet)
    - https://www.aemet.es/xml/municipios/localidad_37274.xml
  - Kyoto Encyclopedia of Genes and Genomes (KEGG)
    - http://rest.kegg.jp/find/genes/shiga+toxin
  - The Cat API ← utilizaremos este como ejemplo
    - https://api.thecatapi.com/v1/images/search

Uso de un servicio web

Autenticación

- Actualmente, casi todos los servicios web requieren algún tipo de autenticación previa
  - Generalmente a través de **OAuth** (Open Authorization), un protocolo de autenticación de APIs
    - O mediante algún sistema más sencillo de registro
  - Complica las invocaciones a la API
  - Mejora la seguridad de los servidores de servicios web

  - No nos vamos a centrar en ello, para más información ver este seminario

REST

Uso de un servicio web en Java

Utilizamos clases de *java.net* y *java.io*, como para acceder a cualquier otro recurso web:

```
URL url = new URL("https://api.thecatapi.com/v1/images/search");
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setRequestMethod("GET");

if (conn.getResponseCode() != 200) {
        throw new RuntimeException("Failed : HTTP error code : "
                        + conn.getResponseCode());
        }

BufferedReader br = new BufferedReader(new InputStreamReader(
                        (conn.getInputStream())));

String output;
                System.out.println("Output from Server .... \n");
                while ((output = br.readLine()) != null) {
                        System.out.println(output);
                }

conn.disconnect();
```

REST

Parseo en Java

- El servicio REST devuelve texto en algún formato
  - Debemos analizarlo para extraer la información que nos interese
  - XML o JSON son formatos muy comunes
- Varias opciones para analizar texto
  - Si tiene una estructura sencilla: *BufferedReader* y *String*
    - Un uso inteligente de *String.split* y *String.replace* suele ser suficiente
  - Si es una estructura compleja:
    - Análisis (parseo) mediante bibliotecas, p. ej.
      - javax.xml.parsers.SAXParser para XML
      - Javax.json.stream.JsonParser para JSON

REST

Uso de un servicio web

Creación de un servicio web en Java

- JAX-RS y anotaciones
- Eclipse + Tomcat + Jersey
- Programas servidor y cliente
- Interfaces

Creación de un servicio REST

JAX-RS

- Para crear un servicio web necesitamos algo más que los objetos de Java para manejo de conexiones
- JAX-RS (Java API for RESTful web services) es una API de Java para crear servicios web tipo REST
  - Jersey (jersey.java.net) es su implementación más estable
- Un objeto java (*POJO* – Plain Old Java Object) se convierte en un recurso web añadiéndole ***anotaciones***
  - Provee información sobre el código, pero no es código
    - Información para la compilación, despligue o ejecución

Creación de un servicio REST

JAX-RS: anotaciones

- @Path indica la ruta relativa a añadir a la URI para acceder a una clase o método
- @GET, @PUT, @POST, @DELETE, @HEAD hacen referencia al tipo de petición HTTP que satisface un método
- @Produces especifica el tipo MIME que retorna (plain, html, json, xml, etc.) un método
  - @Consumes especifica el tipo MIME que requiere un método
- Existen más anotaciones, éstas son sólo las esenciales

Creación de un servicio REST

JAX-RS: anotaciones

- Por ejemplo:

```
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path(“/saludo”);
public String saludar(){ return “Hola”; }
```

- Retornará un mensaje en texto plano que dice “Hola” al acceder a http://host:port/saludo (método GET)

Creación de un servicio REST

Esquema

JAX-RS

Anotaciones JAX-RS

Cliente REST (vía JAX-RS)

Servicio REST

```
ClientConfig conf = new DefaultClientConfig();
Client client = Client.create(conf);

URI uri=UriBuilder
      .fromUri("http://ip:8080/servicePath").build();
WebResource service= client.resource(uri);

System.out.println(service.path("classPath")
      .path("hello").accept(MediaType.TEXT_PLAIN)
      .get(String.class))
```

```
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("hello");
public String saludar(){ return "Hola"; }
```

Creación de un servicio REST

Preparación del entorno

- Descargar **Tomcat** de http://tomcat.apache.org
  - Podríamos descargar otras versiones, pero cuidado con las compatibilidades con Java u otros cambios*
- Descargar **Eclipse EE**
  - A partir de Eclipse IDE 2021-12, descargamos siempre el mismo instalador y luego seleccionamos “Eclipse IDE for Enterprise Java and Web Developers”
- Descargar **Jersey** (https://eclipse-ee4j.github.io/jersey/download.html)
  - Jersey X.Y bundle

* En el laboratorio de informática usaremos Jersey 2.39 y Tomcat 9, con la versión de java 1.8 y Eclipse EE 2018-2020 que están instaladas.
** En casa, cuidado con la compatibilidad entre versiones para Tomcat/Java y Java/Jersey

Creación de un servicio REST

Creación del proyecto

- Crear un nuevo proyecto web:
  - `File/New/Project… > Web/Dynamic Web Project`
- En la carpeta `WebContent/WEB-INF/lib`, incluir todos los jars en las carpetas `jersey/lib`, `jersey/api` y `jersey/ext`
  - Deben colgar directamente, sin las subcarpetas
  - Nota: En las últimas versiones de Eclipse la carpeta `WebContent` puede estar dentro de la carpeta `src`

En http://vis.usal.es/rodrigo/documentos/sisdis/ejemploREST/ se encuentran algunas de las clases y ficheros que vamos a usar de ejemplo

Creación de un servicio REST

Fichero web.xml

- Copiar a `WebContent/WEB-INF` este archivo:
  - http://vis.usal.es/rodrigo/documentos/sisdis/ejemploREST/web.xml
- Modificar dicho archivo:
  - `display-name` debe coincidir con el nombre del proyecto
  - `jersey.config.server.provider.packages` debe tener como valor (`param-value`) la lista de paquetes con recursos REST, separados por punto y coma.
  - `url-pattern` dentro de `servlet-mapping` debe ser la ruta base a partir de la que se ubicarán los recursos REST (puede ser `/*`)

Creación de un servicio REST

Ejemplo de servicio

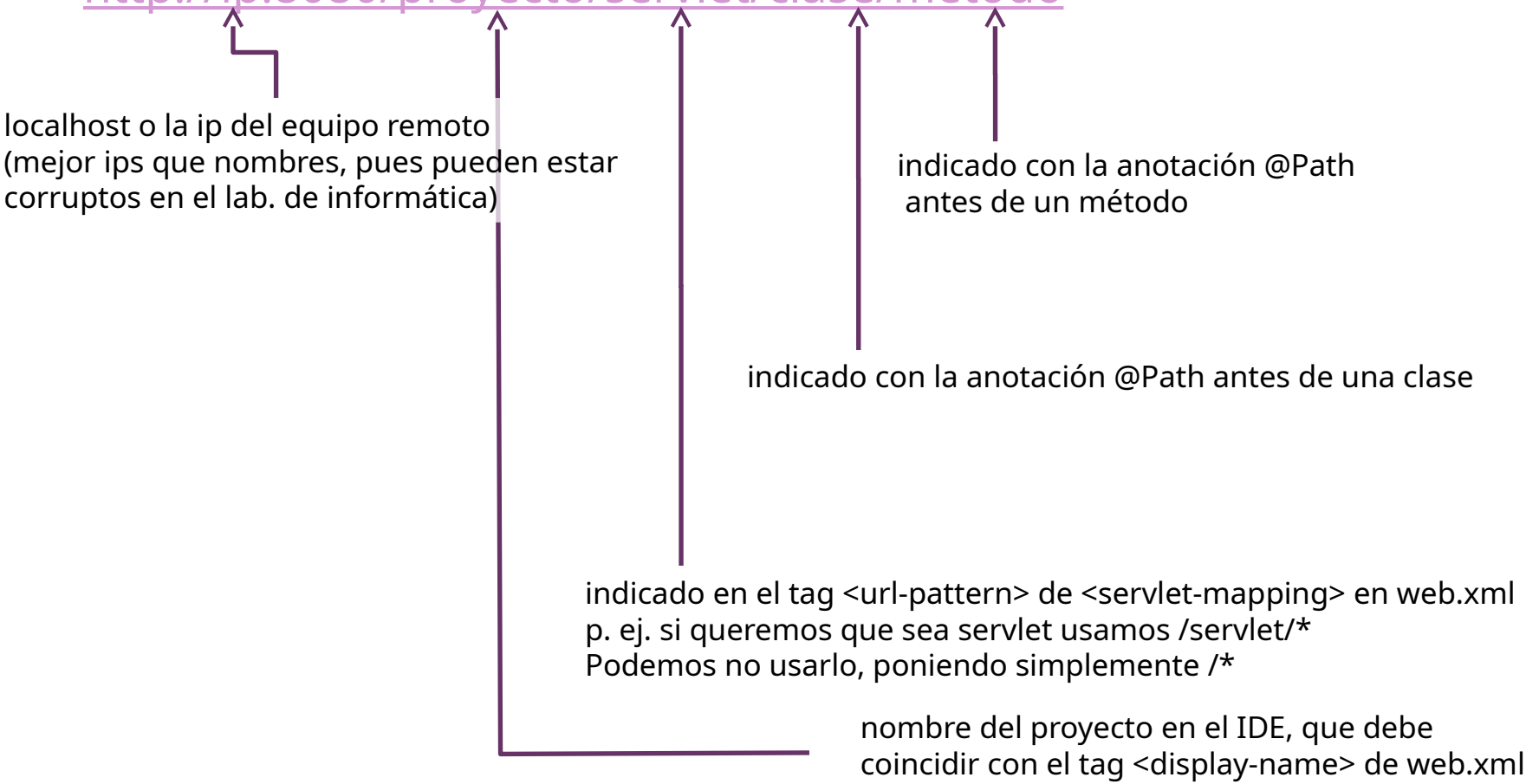
```
@Path("hola")//ruta a la clase
public class HolaMundo
    {
    @GET                          //tipo de petición HTTP
    @Produces(MediaType.TEXT_PLAIN)  //tipo de texto devuelto
    @Path("saludo")               //ruta al método
    public String saludo()        //el método debe retornar String
        {
        return "Hola gente!!";
        }
    }
```

http://vis.usal.es/rodrigo/documentos/sisdis/ejemploREST/HolaMundo.java

Creación de un servicio REST

Ruta del servicio

- http://ip:8080/proyecto/servlet/clase/metodo



Creación de un servicio REST

Arranque del servicio

- Arrancar el servicio: Run/Run As.../Run on Server
  - Especificar Tomcat como servidor en el que arrancarlo
    - Target runtime (o *New...* si no está)
- Errores frecuentes:
  - **java.lang.ClassNotFoundException**: com.sun.jersey.spi.container.servlet.ServletContainer
    - Los jar de Jersey no se han incluido correctamente en WebContent/WEB-INF/lib
  - **com.sun.jersey.api.container.ContainerException**: The ResourceConfig instance does not contain any root resource classes.
    - El parámetro com.sun.jersey.config.property.packages no se ha configurado correctamente en web.xml: debe contener los nombres de los paquetes que contienen clases anotadas.
  - El servidor arranca pero no hay nada en las rutas esperadas
    - El parámetro com.sun.jersey.config.property.packages no se ha configurado correctamente en web.xml: debe contener los nombres de los paquetes que contienen clases anotadas.
    - Revisar los @Path, y los tags <display-name> y<servlet-mapping> en web.xml

Creación de un servicio REST

Ejemplo de cliente

```
public class Test {
        public static void main(String args[])
                {
                Client client=ClientBuilder.newClient();
                URI uri=UriBuilder.fromUri("http://localhost:8080/pruebasREST").build();

                WebTarget target = client.target(uri);


                System.out.println(target.path("rest").path("hola").path("saludo").
                                        request(MediaType.TEXT_PLAIN).get(String.class));
                }
}
```

Se ejecuta como una aplicación Java normal

http://vis.usal.es/rodrigo/documentos/sisdis/ejemploREST/Test.java

Ejercicio

- Crear un servicio REST *hello* mediante Eclipse, Tomcat y Jersey.
  - Iniciar en la máquina local y probar accesos de clientes
    - Desde un navegador y desde java
    - Desde la máquina local y desde otras máquinas

Creación de un servicio REST

Paso de argumentos

- Paso de argumentos: anotación @**QueryParam**:

```
@Path(”calculator")
public class Calculator
 {
 @Path(“sq")
 @GET
 @Produces(MediaType.TEXT_PLAIN)
 public String square(@DefaultValue("2") @QueryParam(value="num") long num)
  {
  return ""+num*num;
  }
 }
```

- Desde URL http://hostname:port/calculator/sq?num=3

- Desde Java
  - service.path("calculator/sq").queryParam("num", ””+3).request
    (MediaType.TEXT_PLAIN).get(String.**class**)

Creación de un servicio REST

Llamadas asíncronas

- Tratamiento de la respuesta mediante callbacks*

```
target.path("http://ejemplo.com/servicio").request().async().get(
    new InvocationCallback<Response>(){
    @Override
    public void completed(Response response)
        {
        System.out.println("Code " + response.getStatus() + " received.");
        System.out.println("Response: " + response.readEntity(String.class));
        }

    @Override
    public void failed(Throwable throwable)
        {
        System.out.println("Invocation failed.");
        throwable.printStackTrace();
        }

    });
System.out.println("Sigo a lo mío");
```

*Más información en https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/async.html

Creación de un servicio REST

Minimización de interfaces

- Respecto al uso de argumentos y el retorno de objetos, un buen diseño de un sistema distribuido minimiza las interfaces
  - Suponen una carga en el tráfico de red
    - Y más si hay que convertirlos a XML
  - Incrementan el riesgo de errores
    - Interpretaciones equivocadas de la API
    - Las clases tienen que estar disponibles por los clientes
    - Etc.
  - Muchos objetos son evitables con un uso inteligente de String

Creación de un servicio REST

Ciclo de vida de los objetos

- En Jersey, los objetos tienen un ciclo de vida '*per-request*'
  - Cada clase que se ofrece como recurso se instancia con cada nueva petición y se destruye al terminar dicha petición
    - Esto impide mantener objetos que varían su estado a lo largo del tiempo (a través de distintas peticiones)
  - Solución:
    - Utilizar la anotación @**Singleton** para la clase
      - Así, la clase se instancia una vez por aplicación web, y permanece instanciada hasta que se apague o reinicie el servicio

Ejercicio

- Crear un servicio REST *calculator* que permita realizar potencias cuadradas (*sq*) y sumas de dos elementos (*add*)
  - Obtendrá mediante parámetros el número a elevar al cuadrado y los dos números a sumar (todos enteros)
  - Retornará el resultado como una cadena de texto

- Añadir una tercera función *stack*(int *n*) que sume el valor *n* a una variable interna del servicio que comienza en *0*

REST

Tutoriales

- http://www.vogella.com/articles/REST/article.html
  - Preparación básica para trabajar con Jersey+Tomcat+Eclipse
- https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest3x
- https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest
  - Manual completo de Jersey 3x o 2x, en especial:
    - Paso de argumentos (cap 3.2)
    - Ciclo de vida de los recursos (3.4)

Carrera 100m lisos

Carrera 100m lisos

Servicio

- Crear un servicio REST mediante una clase `Carrera100`
  - El servicio se llamará **carrera100** y aceptará 4 atletas
  - Mantendrá información sobre
    - Número de atletas inscritos en la carrera
    - Tiempo de inicio de la carrera y de llegada de cada atleta
  - Ofrecerá los métodos
    - **reinicio**: pone los tiempos y los atletas inscritos a cero
    - **preparado**: detiene al atleta que lo llama hasta que todos los atletas estén preparados
    - **listo**: detiene al atleta que lo llama hasta que todos los atletas estén listos. Una vez estén todos listos, la carrera empieza
    - **llegada**(**dorsal**): guarda el tiempo de llegada del atleta y retorna el tiempo obtenido por el atleta.
    - **resultados**: retorna una cadena con algún formato que muestre los resultados de la carrera

Carrera 100m lisos

Cliente

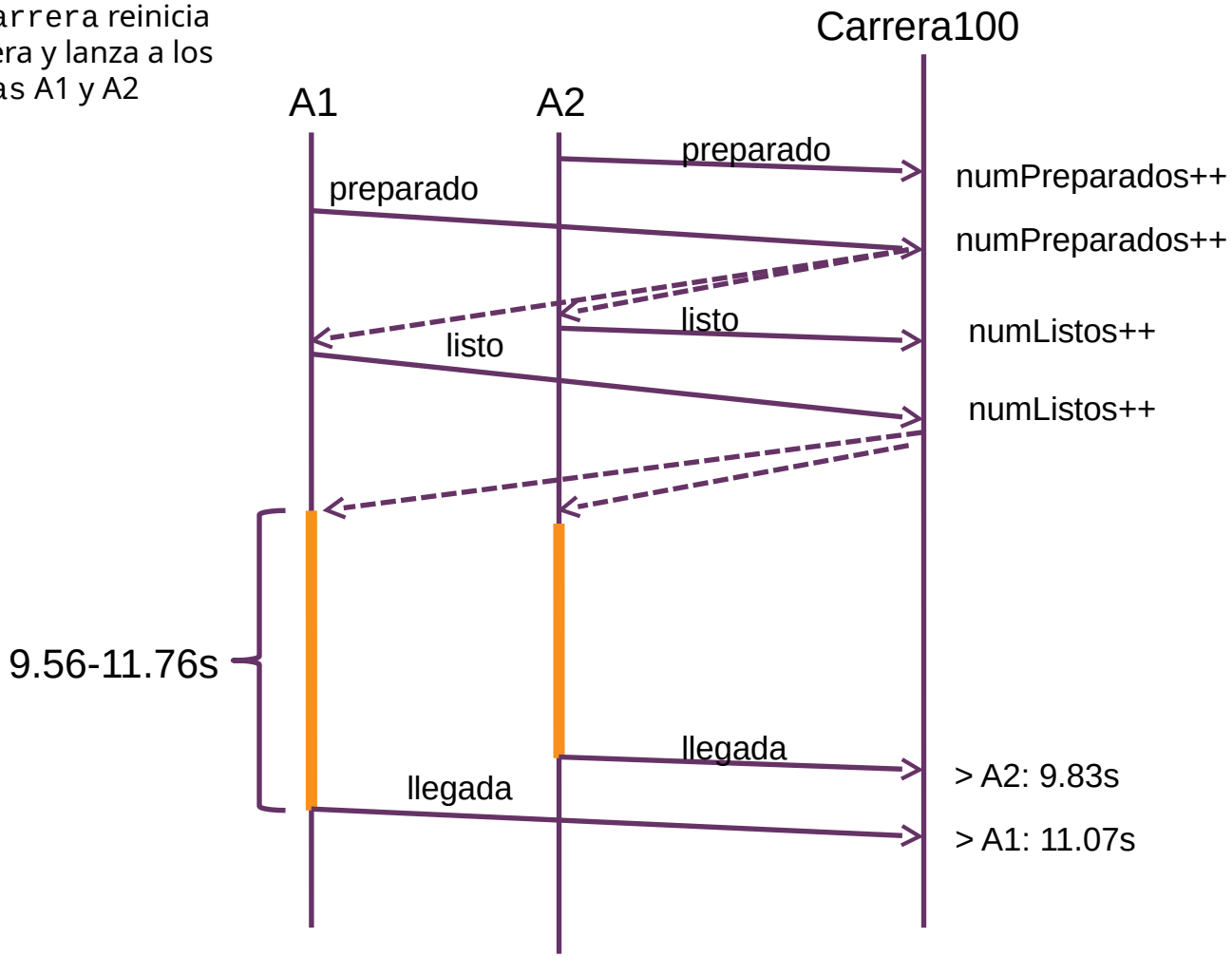
- La clase `Atleta` será un hilo (`Thread`) que:
  - Se construirá con un determinado dorsal
  - Durante su ejecución
    1. Invoca `carrera100/preparado`
    2. Invoca `carrera100/listo *`
    3. Corre (duerme entre 9.56 y 11.76s)
    4. Invoca `carrera100/llegada?dorsal=midorsal`
- Para hacer una carrera puede haber una clase `MainCarrera`
  1. Invoca `carrera100/reinicio`
  2. Crea 4 Atletas y los pone a correr
  3. Invoca `carrera100/resultados`

* Podríamos eliminar esta invocación y servicio, basta con el servicio `preparado`, ¡pero hay que tener sentido del drama!

Carrera 100m lisos

Ejemplo con 2 procesos

`MainCarrera` reinicia la carrera y lanza a los `Atletas` A1 y A2



Carrera 100m lisos

Despliegue

- Ejecutar el servicio y los atletas en el mismo ordenador
- Probar con 2 ordenadores
  - En uno corre el servicio y dos atletas
  - En el otro corren los otros dos atletas
- Probar con 3 ordenadores, con 6 atletas
  - En cada uno corren dos atletas
  - En uno de ellos corre el servicio

Carrera 100m lisos

Despliegue: determinar IP del servidor

- Para que los clientes sepan dónde está
  - `/sbin/ifconfig`
  - `/sbin/ifconfig | grep 'inet addr:' | grep -v '127.0.0.1' | cut -d: f2 | awk '{print $1}'`
    - Para extraer los números de la ip
    - O más sencillo, directamente con `hostname -I`

Carrera 100m lisos

Despliegue: reparto de clases

- **Básico**:
  - Almacenar las clases en Z:
    - Estarán disponibles en todos los ordenadores si nos conectamos con el mismo usuario
- **Avanzado**:
  - Pensando en otros sistemas donde no tengamos un servicio distribuido de archivos
  - Podemos generar un script de envío remoto mediante `ssh/scp`
    - Ver los scripts `lanzar.sh` y `shareKeys.sh` en http://vis.usal.es/rodrigo/documentos/sisdis/scripts
- **Pro**:
  - Podemos generar un .jar con las clases y bibliotecas necesarias y enviarlas mediante scripts/ssh
    - Si usamos Tomcat, podemos generar un .war y almacenarlo en la carpeta *webapps*

Carrera 100m lisos

Despliegue: ejecución

- El servidor se arranca inicialmente
  - **Básico**: mediante Eclipse, con `Run as.../Run on Server`
  - **Avanzado**: usar el proyecto .war del despliegue Pro
  - Pro: crear un demonio que arranque con el ordenador
- Luego arrancamos los clientes
  - **Básico**: a través de Eclipse (requiere arrancar Eclipse en todos los ordenadores)
  - **Avanzado**: ejecutarlos desde consola, localmente (requiere acceso físico a todos los ordenadores)
  - **Pro**: ejecutarlos desde consola, remotamente (todo se hace desde un solo ordenador)
    - Podemos usar los scripts vistos en el reparto de clases

Carrera 100m lisos

Coordinación y tiempos

- Probad qué pasa si los Atletas no esperan a las órdenes de ‘preparados’ y ‘listos’, y empiezan a correr en cuanto pueden
  - En distintos despliegues
- Reflexionad y/o probad con diferentes medidas de tiempos
  - En `Carrera100` o por los propios Atletas
  - En distintos despliegues
  - De forma relativa (“he tardado *t_final* menos *t_inicial*”)
    - Obteniendo ellos el *t_inicial*
    - Tomando *t_inicial* de la carrera
  - De forma absoluta (“he llegado en *t_final*”)

Carrera 100m lisos

Análisis

- ¿Qué posibles fallos encuentras en el sistema que has implementado?
  - Relativos a los tiempos
  - Relativos a la coordinación
  - Relativos a posibles fallos de proceso
  - Relativos a posibles fallos de comunicación
- ¿Se te ocurren mejoras posibles para el sistema?